DOI:
10.1016/j.tcs.2016.05.028

Document Version
Peer reviewed version

Accepted Manuscript

Binary block order Rouen transform

Jacqueline W. Daykin, Richard Groult, Yannick Guesnet, Thierry Lecroq, Arnaud Lefebvre et al.

PII: S0304-3975(16)30164-5
DOI: http://dx.doi.org/10.1016/j.tcs.2016.05.028
Reference: TCS 10777

To appear in: *Theoretical Computer Science*

Received date: 7 August 2015
Revised date: 3 May 2016
Accepted date: 19 May 2016

Please cite this article in press as: J.W. Daykin et al., Binary block order Rouen transform, *Theoret. Comput. Sci.* (2016), http://dx.doi.org/10.1016/j.tcs.2016.05.028

Highlights

- Novel twin binary Burrows-Wheeler type transforms are introduced.
- The transforms are defined for Lyndon-like B-words which apply binary block order.
- We call this approach the B-BWT Rouen Transform.
- These bijective Rouen transforms and inverses are computed in linear time.
- Preliminary experimental results indicate potential value of binary transforms.

Binary Block Order Rouen Transform☆

Jacqueline W. Daykin[a,b,d], Richard Groult[c,d], Yannick Guesnet[d], Thierry Lecroq[d], Arnaud Lefebvre[d], Martine Léonard[d], Élise Prieur-Gaston[d]

[a] *Department of Computer Science, Royal Holloway, University of London, UK*
[b] *Department of Informatics, King's College London, UK*
[c] *Modélisation, Information et Systèmes (MIS), Université de Picardie Jules Verne, Amiens, France*
[d] *Normandie Université, Université de Rouen-Normandie, LITIS EA 4108, Normastic FR CNRS 3638, 76821 Mont-Saint-Aignan, France*

Abstract

We introduce bijective Burrows-Wheeler type transforms for binary strings [1]. The original method by Burrows and Wheeler [BW94] is based on lexicographic order for general alphabets, and the transform is defined to be the last column of the ordered BWT matrix. This new approach applies binary block order, B-order, which yields not one, but twin transforms: one based on Lyndon words, the other on a repetition of Lyndon words. These binary B-BWT transforms are constructed here for B-words, analogous structures to Lyndon words. A key computation in the transforms is the application of a linear-time suffix-sorting technique, such as [KA03, KS03, KSB06, NZC09], to sort the cyclic rotations of a binary input string into their B-order. Moreover, like the original lexicographic transform, we show that computing the B-BWT inverses is also achieved in linear time by using straightforward combinatorial arguments.

Keywords: algorithm, bijective, binary alphabet, block order, Burrows-Wheeler Transform, B-word, data clustering, inverse transform, lexicographic order, linear, Lyndon word, string, suffix array, suffix-sorting, word

1. Introduction

The Burrows-Wheeler Transform (BWT) is a well-studied and useful text transformation scheme which was introduced by Burrows and Wheeler in 1994 [BW94]. The transform is constructed by arranging all cyclic rotations, or conjugates, of a string into a matrix in lexicographic order and extracting the rightmost column – hence a permutation of the input. Notably the transform

☆ *This paper is in honour of eminent stringologist Professor Bill Smyth: our colleague, friend and fellow life traveller. Bill is an inspiration to us all!*

[1] These twin transforms originated in problem solving sessions of the Rouen 2012 String-Masters workshop, hence the name Rouen Transform.

has the property that it tends to cluster occurrences of the same letter in the input string into repetitions (runs), and moreover, that it is reversible. Due to this data-clustering property, the Burrows-Wheeler Transform finds wide-ranging applications in: preprocessing for data compression (it is also called block-sorting compression) and it is core to the bzip2 family of text compressors; bioinformatics and sequence processing [BCRS13, MRRS05]; and text indexing [FM00] – indeed the versatility of the transform is increasing to include, for instance, multimedia information retrieval [ABM08].

During the last decade, further research into the combinatorial properties of the BWT has led to related observations and efficient time and space implementations, including [CDP05, CGKL13, Kär07, KKP12, SLLM09], along with a range of bijective variants such as the sort transform [GS12, Kuf09].

In this paper we continue more recent research into Burrows-Wheeler type transforms based on non-lexicographic orders, thus contributing towards a taxonomy of BWT-type transforms. In [DS14] the V-BWT transform is constructed which is derived from V-order on an arbitrary alphabet; in [DW14] the D-BWT degenerate transform is introduced for indeterminate strings, namely strings of subsets of an alphabet. We propose here the binary B-BWT Rouen Transform, based on binary block B-order, to achieve text transformations for binary strings: preliminary experiments indicate potential gains of the binary Rouen Transform over the original lexicographic BWT, encouraging further practical research.

Space saving techniques with the classic BWT have been achieved by first factoring the input string into Lyndon words in linear time [CFL58, Day11, Duv83, Lot83, Smy03]. Similarly, in Algorithm 1, we efficiently factor the input string into B-words, namely analogues of Lyndon words defined using B-order [DD96, DD03] – we achieve transforms here for the case of B-words.

To sort the rotations of a binary string into a BWT matrix in B-order, we will apply a linear-time suffix-sorting algorithm, such as that of Kärkkäinen and Sanders [KS03], Kärkkäinen, Sanders and Burkhardt [KSB06], Ko and Aluru [KA03], or Nong, Zhang, and Chan [NZC09].

To our knowledge these are the first Burrows-Wheeler type transforms defined specifically for a binary alphabet; furthermore, the ordering technique applied, B-order, is based on blocks or runs of bits, and so is itself tuned to clustering – see Definition 2.1. However, it is interesting to note that the concept of modifying order was proposed in [CT98], soon after the transform was first introduced. In particular, they showed that re-ordering the characters of the alphabet provides superior compression than using their original order: the main difference with the original order is the grouping of the vowels at the beginning of the alphabet. Further, the gain for this alphabet re-ordering algorithm (with respect to the original order) for a BWT-based compressor was later shown in [AT05].

1.1. Notation

We introduce some standard notation; for further automata and stringology terminology, theory, algorithms and data structures see [CHL07, Smy03].

A ***string***, or equivalently ***word***, is a sequence of elements, letters or symbols over a set Σ, an ***alphabet***. Throughout this paper the given alphabet is assumed to be binary, $\Sigma = \{0, 1\}$, and the binary elements are ***bits***; the binary strings may be encoded as integers thus inducing the alphabet $\Sigma' = \{0, 1, 2, \ldots\}$. All strings are written in mathbold: $\boldsymbol{x}$, $\boldsymbol{w}$, and so on; $\boldsymbol{\varepsilon}$ denotes the empty string.

Let $\boldsymbol{x}$ be the binary string $\boldsymbol{x} = \boldsymbol{x}[1 \ldots n] = x_1 x_2 \ldots x_n$, where each $x_i \in \{0, 1\}$, and the ***length*** $|\boldsymbol{x}| = n$. Hence the i-th symbol of a string $\boldsymbol{x}$ is denoted by $\boldsymbol{x}[i]$, or simply x_i. We denote by $\boldsymbol{x}[i \ldots j]$, or $x_i \ldots x_j$, the ***substring*** of $\boldsymbol{x}$ that starts at position i and ends at position j. A string $\boldsymbol{w}$ is a substring, or ***factor***, of $\boldsymbol{x}$ if $\boldsymbol{x} = \boldsymbol{uwv}$, where $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$; specifically, a string $\boldsymbol{w} = w_1 \ldots w_m$ is a substring of $\boldsymbol{x} = x_1 \ldots x_n$ if $w_1 \ldots w_m = x_i \ldots x_{i+m-1}$ for some i. Words $\boldsymbol{x}[1 \ldots i]$ are called ***prefixes*** of $\boldsymbol{x}$, and words $\boldsymbol{x}[i \ldots n]$ are called ***suffixes*** of $\boldsymbol{x}$. The prefix $\boldsymbol{u}$ (respectively suffix $\boldsymbol{v}$) is a ***proper*** prefix (suffix) of a word $\boldsymbol{x} = \boldsymbol{uv}$ if $\boldsymbol{x} \neq \boldsymbol{u}$ ($\neq \boldsymbol{v}$).

If we can write $\boldsymbol{x} = \boldsymbol{uv} = \boldsymbol{wu}$ for some nonempty $\boldsymbol{u}$, then we say that $\boldsymbol{x}$ has ***border*** $\boldsymbol{u}$; if no such $\boldsymbol{u}$ exists, then $\boldsymbol{x}$ is said to be ***border-free*** (or ***unbordered***). If $\boldsymbol{x} = \boldsymbol{u}^k$ for some nonempty $\boldsymbol{u}$ and some integer $k > 1$, we say that $\boldsymbol{x}$ is a ***repetition*** with ***seed*** $\boldsymbol{u}$; otherwise, we say that $\boldsymbol{x}$ is ***primitive***. If a string $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is said to be a ***rotation*** (cyclic shift) or ***conjugate*** of $\boldsymbol{x}$.

The subject of this paper requires careful study of rotations of strings with associated notation. For a binary string $\boldsymbol{x}$, we let $\boldsymbol{x}^+$, $\boldsymbol{x}_0^+$, $\boldsymbol{x}_1^+$ each denote a clockwise rotation of (any) one bit, a 0 bit, and a 1 bit respectively. Likewise, let $\boldsymbol{x}^-$, $\boldsymbol{x}_0^-$, $\boldsymbol{x}_1^-$ denote an anti-clockwise rotation of (any) one bit, a 0 bit, and a 1 bit respectively. For sequences of bit rotations, let $(\boldsymbol{x}^+)^k$ denote k clockwise rotations, and similarly for $(\boldsymbol{x}_0^-)^k$ and so on. For instance, if $\boldsymbol{x} = 1010010$, then $\boldsymbol{x}^+ = 0100101$ and $\boldsymbol{x}^- = 0101001$; further, since it is known that the rightmost bit of $\boldsymbol{x}$ is 0, then we can apply $\boldsymbol{x}_0^-$, giving $\boldsymbol{x}_0^- = \boldsymbol{x}^- = 0101001$. Unless specified otherwise, a rotation will be assumed to be anti-clockwise.

We now identify four disjoint sets of rotations of $\boldsymbol{x}$ which arise in the B-BWT binary transform algorithms:

- $\mathcal{R}_0^u$ - unbordered rotations commencing with 0 and ending with 1
- $\mathcal{R}_1^u$ - unbordered rotations commencing with 1 and ending with 0
- $\mathcal{R}_0^b$ - bordered rotations commencing with 0 and ending with 0
- $\mathcal{R}_1^b$ - bordered rotations commencing with 1 and ending with 1.

Note that with our focus on rotating single bits, we use a weak form of border-free here: for instance, if only the first and last bits are known, the string $001 \ldots 001$ would be considered border-free.

We may encode a non-empty binary string $\boldsymbol{x}$ as the string of integers $\hat{\boldsymbol{x}} = Z_0 N_1 Z_1 N_2 Z_2 \ldots N_r Z_r$, where Z_0 is the number of 0's at the start of $\boldsymbol{x}$, then N_1 is the number of following 1's, and so on. For example, if $\boldsymbol{x} = 0111001111$

then $r = 2$ and $\hat{\boldsymbol{x}} = 13240$. In general $Z_0, Z_r \geq 0$ but $N_1, Z_1, \ldots, N_r \geq 1$. Given an encoding $\hat{\boldsymbol{x}} = Z_0N_1Z_1N_2Z_2 \ldots N_rZ_r$, let $\hat{N}(\boldsymbol{x}) = \hat{N} = N_1N_2 \ldots N_r$, and similarly $\hat{Z}(\boldsymbol{x}) = \hat{Z} = Z_0Z_1Z_2 \ldots Z_r$. Notation such as $N_i(\boldsymbol{x})$ (respectively $Z_i(\boldsymbol{x})$) would indicate the ith N (respectively Z) value of an encoded string $\hat{\boldsymbol{x}}$. The maximal substrings 0^{Z_i} and 1^{N_j} of $\boldsymbol{x}$ are called ***blocks***, or in stringology they are usually called ***runs***; the terminology N block (Z block) will mean a block of 1's (0's).

Given the encoding $\hat{\boldsymbol{x}} = Z_0N_1Z_1N_2Z_2 \ldots N_rZ_r$ of a border-free string $\boldsymbol{x}$, if a rotation(s) of $\boldsymbol{x}$ causes $\boldsymbol{x}$ to become bordered, that is by rotating the bits of a prefix/suffix of a block, then that block becomes a ***split block*** (otherwise a ***non-split*** or ***complete block***); for instance, if $\boldsymbol{x} = 010010111$, where $\hat{\boldsymbol{x}} = Z_0N_1Z_1N_2Z_2N_3Z_3 = 1121130$, then $(\boldsymbol{x}_1^-)^2 = 110100101$ and the block associated with $N_3 = 3$ has become a split block. We call maximal substrings in $\boldsymbol{x}$ of the form $1^{N_i}0^{Z_i}$ a ***block pair***.

Let the ***weight*** $\omega\boldsymbol{x}$ of a binary string $\boldsymbol{x}$ be the total number of 1's in $\boldsymbol{x}$, that is, $\omega\boldsymbol{x} = x_1 + x_2 + \cdots + x_n$; thus all rotations in the set of cyclic rotations, or conjugates, of $\boldsymbol{x}$ have the same weight ω.

This paper is also concerned with methods for totally ordering strings. If a string $\boldsymbol{x}$ is less than a string $\boldsymbol{y}$ in ***lexorder*** (lexicographic order), we write $\boldsymbol{x} < \boldsymbol{y}$ and $\boldsymbol{y} > \boldsymbol{x}$. The next section discusses a binary string ordering technique.

2. Binary Block Order (*B*-order)

We start with the basic definitions and computational issues for working with B-order in Section 2.1, followed by some combinatorial results in Section 2.2 which will be applied when computing a transform and inverse.

2.1. Block Order Preliminaries

We introduce here the method for totally ordering binary strings which will be used in Sections 3 & 4 for constructing the B-BWT matrices followed by extracting the transforms.

Definition 2.1. *[DD96] B-*order*(**Block total order** $\ll$) of words of length n. Given $\boldsymbol{x} = x_1x_2 \ldots x_n$ with encoding $\hat{\boldsymbol{x}} = Z_0N_1Z_1N_2Z_2 \ldots N_rZ_r$ and $\boldsymbol{y} = y_1y_2 \ldots y_n$ with encoding $\hat{\boldsymbol{y}} = Z_0'N_1'Z_1'N_2'Z_2' \ldots N_s'Z_s'$, where each $x_i, y_j \in \{0, 1\}$ and $|\boldsymbol{y}| = |\boldsymbol{x}|$, then $\boldsymbol{x} \ll \boldsymbol{y}$ if*
(i) $\omega\boldsymbol{x} < \omega\boldsymbol{y}$, or
(ii) $\omega\boldsymbol{x} = \omega\boldsymbol{y}$ and there is a least i in $1 \leq i \leq min\{r, s\}$ with $N_i \neq N_i'$ and then $N_i > N_i'$, or
(iii) $\omega\boldsymbol{x} = \omega\boldsymbol{y}$ and $r = s$ and $N_i = N_i'$ for $1 \leq i \leq r$, and there is a least j in $0 \leq j < r$ with $Z_j \neq Z_j'$ and then $Z_j < Z_j'$.

That is, in B-order, strings are ordered first by weight, then by lexorder on the sizes of blocks of 1's (over the alphabet $\{1 > 2 > 3 > \cdots\}$) and finally by lexorder on the sizes of blocks of 0's (over the alphabet $\{0 < 1 < 2 < 3 < \cdots\}$).

We will describe sorting using this ordering technique as B-***sorting*** or B-***ordering*** strings.

A primitive operation is the comparison of two strings $\boldsymbol{x}$,$\boldsymbol{y}$ in B-order. This can be achieved by straightforward bit comparisons; similarly, following the linear computation of $\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}$, for instance by Run Length Encoding, applying linear lexordering to the weights ω, the $\hat{N}$ values and the $\hat{Z}$ values as necessary, we get:

Claim 2.2. *The comparison of two strings $\boldsymbol{x}$, $\boldsymbol{y}$ in B-order, that is to determine $\boldsymbol{x} \ll \boldsymbol{y}$, can be achieved in linear time.*

In this paper we will demonstrate twin transforms for the case of B-words which are defined as follows:

Definition 2.3. *[DD03] A binary primitive string $\boldsymbol{x}$ is a* B-**word** *if it is the unique minimum in B-order among the cyclic rotations of $\boldsymbol{x}$.*

Hence B-words are analogues of Lyndon words – specified as lexicographically least in their set of conjugates [CFL58, Lot83] – while defined with B-order rather than the original lexorder; another analogue of Lyndon words, defined with V-order, is given in [DD03, DS14].

Example 2.4. *Consider the cyclic rotations of the string* 110100.
In B-order: $110100 \ll 011010 \ll 001101 \ll 100110 \ll 010011 \ll 101001$.
Therefore the given string 110100 *is a B-word.*

Furthermore: 0, 1, and strings (block pairs) of the form 1^j0^k with $j, k > 0$ are B-words; any other string $\boldsymbol{x}$ which is a B-word is in the set $\mathcal{R}_1^u$ with $Z_0(\boldsymbol{x}) = 0$. Note that B-words share similar properties to Lyndon words such as being primitive, border-free and must start with the largest N value; hence we denote N_1 as *max*. We now give a characterization:

So let $\boldsymbol{x} = x_1x_2 \dots x_n \in \mathcal{R}_1^u$ be a B-word; then $\hat{\boldsymbol{x}} = Z_0N_1Z_1N_2Z_2 \dots N_rZ_r$ with $Z_0(\boldsymbol{x}) = 0$. If $\hat{N}(\boldsymbol{x}) = (N_1 \dots N_p)^k$ then let $\zeta(\boldsymbol{x}) = \zeta_1\zeta_2 \dots \zeta_k$ with $\zeta_i = Z_{(i-1)p+1}Z_{(i-1)p+2} \dots Z_{(i-1)p+p}$. Notation such as $\zeta_i(\boldsymbol{x})$ would indicate the ith ζ value of an encoded string $\hat{\boldsymbol{x}}$.

Theorem 2.5. *A string $\boldsymbol{x} \in \mathcal{R}_1^u$ is a B-word if and only if either $\hat{N}(\boldsymbol{x})$ forms a Lyndon word over the alphabet $\{1 > 2 > 3 > \cdots\}$, or, $\hat{N}(\boldsymbol{x})$ forms a repetition of a Lyndon word over the alphabet $\{1 > 2 > 3 > \cdots\}$ and $\zeta(\boldsymbol{x})$ forms a Lyndon word over the alphabet $\{1^p < 1^{p-1}2 < 1^{p-1}3 < \cdots\}$.*

Proof. ($\Rightarrow$) Suppose first that $\boldsymbol{x}$ is a B-word. Then Definition 2.1 ensures that $\hat{N}(\boldsymbol{x})$ is either a Lyndon word over $\{1 > 2 > 3 > \cdots\}$ (ii) or a repetition of Lyndon words over the alphabet $\{1 > 2 > 3 > \cdots\}$.

Consider the case that $\hat{N}(\boldsymbol{x})$ is a repetition of Lyndon words over $\{1 > 2 > 3 > \cdots\}$. If $\zeta(\boldsymbol{x})$ does not form a Lyndon word over the alphabet $\{1^p < 1^{p-1}2 < 1^{p-1}3 < \cdots\}$ then there exists a rotation $\boldsymbol{r}$ of $\boldsymbol{x}$ such that $\zeta(\boldsymbol{r}) \ll \zeta(\boldsymbol{x})$

which contradicts Definition 2.3 stating that $\boldsymbol{x}$ is strictly smaller than all of its rotations.

($\Leftarrow$) Suppose now that $\hat{N}(\boldsymbol{x})$ and $\zeta(\boldsymbol{x})$ satisfy either of the stated Lyndon properties for a string $\boldsymbol{x} \in \mathcal{R}_1^u$. Then by Definition 2.1, $\boldsymbol{x}$ must precede any of its rotations in B-order, including those with split blocks, and hence forms a B-word. □

Example 2.6. *Let* $\boldsymbol{x} = 11101001110010$ *then* $\hat{N}(\boldsymbol{x}) = 3131 = (31)^2$, $\hat{Z}(\boldsymbol{x}) = 01221$, $\zeta(\boldsymbol{x}) = 12 \cdot 21$ *with* $\zeta_1(\boldsymbol{x}) = 12 < 21 = \zeta_2(\boldsymbol{x})$*:* $\hat{N}(\boldsymbol{x})$ *is a repetition of Lyndon words and* $\zeta(\boldsymbol{x})$ *is a Lyndon word over their respective alphabets thus* $\boldsymbol{x}$ *is a* B*-word. Now let* $\boldsymbol{x'} = 11100101110100$ *which is a conjugate of* $\boldsymbol{x}$ *then* $\hat{N}(\boldsymbol{x'}) = 3131 = (31)^2$, $\hat{Z}(\boldsymbol{x'}) = 02112$, $\zeta(\boldsymbol{x'}) = 21 \cdot 12$ *with* $\zeta_1(\boldsymbol{x'}) = 21 > 12 = \zeta_2(\boldsymbol{x'})$*:* $\hat{N}(\boldsymbol{x'})$ *is a repetition of Lyndon words but* $\zeta(\boldsymbol{x'})$ *is not a Lyndon word over their respective alphabets thus* $\boldsymbol{x'}$ *is not a* B*-word. Note that the border-free property of the Lyndon word* $\zeta(x)$ *is given by the words* ζ_i *and not their individual letters.*

We will refer to the first case in Theorem 2.5 as ***type primitive*** B***-words,*** B***-prim***, and the second as ***type repetition*** B***-words,*** B***-rep*** – these two types of words yield two different methods for computing and inverting transforms to recover the input.

We will require all B-ordered rotations of a string for computing a transform, illustrated by:

Example 2.7. *Consider the* B*-BWT matrix* $M_{\boldsymbol{x}}$ *of cyclic rotations for* $\boldsymbol{x} = 110110010$*, ordered into block order, giving* $M_{\boldsymbol{x}}^B$*:*

1	1	0	1	1	0	0	1	0
0	1	1	0	1	1	0	0	1
1	1	0	0	1	0	1	1	0
0	1	1	0	0	1	0	1	1
1	0	1	1	0	1	1	0	0
0	1	0	1	1	0	1	1	0
0	0	1	0	1	1	0	1	1
1	0	1	1	0	0	1	0	1
1	0	0	1	0	1	1	0	1

In this example with $\boldsymbol{x}$ a B-prim, we have $\hat{N}(\boldsymbol{x}) = 221$, a Lyndon word (using $>$ rather than the usual $<$); hence, by Definition 2.1(ii), all rotations with non-split N blocks occur after $\boldsymbol{x}$ in $M_{\boldsymbol{x}}$. Again by Definition 2.1(ii), rotations in $M_{\boldsymbol{x}}$ with split N blocks will also occur after $\boldsymbol{x}$. Note that $\hat{Z}(\boldsymbol{x}) = 0121$ is not then required to be a Lyndon word (although it is using $<$), since the B-word criteria of $\boldsymbol{x}$ is guaranteed here by the Lyndon property of $\hat{N}$.

Note that to test if a string $\boldsymbol{x}$ which starts with 1 is a B-word, by Lyndon properties, we need only look at rotations of $\boldsymbol{x}$ in the set $\mathcal{R}_1^u$, as any rotation

starting with a $Z \neq 0$ block has a prefix of 0's of length greater than the required $Z_0(\boldsymbol{x}) = 0$, satisfying Definition 2.1(iii). Similarly, for rotations of $\boldsymbol{x}$ in the set $\mathcal{R}_1^b$ which arise from split N blocks, Definition 2.1(ii) applies.

Furthermore, we can factor an arbitrary binary string $\boldsymbol{x}$ of length n bits into B-words in linear time using a modification of Duval's $O(n)$ Lyndon decomposition algorithm [Duv83, Day11]: first compute $\hat{\boldsymbol{x}}$ in linear time using Run Length Encoding; then, in tandem, factor the N's into Lyndon words using $>$, while for any detected repetitions of Lyndon words, factor the Z components into Lyndon words using $<$, taking the B-words to be those factors with maximal length – see Algorithm 1. So, without incurring any computational inefficiency, we will assume that the input to the transform is a B-word.

Algorithm 1 Factor a binary string $\boldsymbol{x}$ into B-words in $O(n)$-time.

procedure COMPUTE-BF($\boldsymbol{x}$)
  $F \leftarrow \emptyset$;
  Compute $\hat{\boldsymbol{x}} = Z_0 N_1 Z_1 N_2 Z_2 \dots N_r Z_r$;
    ▷ compute the $\hat{\boldsymbol{x}}$ encoding using Run Length Encoding

  Compute $NLF(N_1 N_2 \dots N_r) = \boldsymbol{n_1 n_2 \cdots n_s}$;
    ▷ compute the Lyndon factorization of N components using $>$

  for each repetition $(\boldsymbol{n_p} = \boldsymbol{n_{p+1}} = \cdots = \boldsymbol{n_q})$ detected during factoring **do**
    Compute $\zeta LF(\boldsymbol{n_p} = \boldsymbol{n_{p+1}} = \cdots = \boldsymbol{n_q}) = \boldsymbol{\theta_1 \theta_2 \cdots \theta_t}$;
      ▷ compute Lyndon factorization of Z components corresponding
      ▷ to $(\boldsymbol{n_p} = \cdots = \boldsymbol{n_q})$ using $<$

    for $i \leftarrow 1$ to t **do**
      add end index of θ_i to F
    end for
  end for
  return F
end procedure

Note that due to the maximality property of the Lyndon factorization, the factorization into B-words will likewise be maximal and thus unique. To formalize this, we apply the following lemma with the terminology: an *UMFF* is a unique maximal factorization family *FF*, where an FF is a subset $\mathcal{W} \subseteq \Sigma^+$ which permits, for every nonempty string $\boldsymbol{x}$ on Σ, a factorization of $\boldsymbol{x}$ over $\mathcal{W}$.

Lemma 2.8. *(The $\boldsymbol{xyz}$ Lemma [DD03]). An FF $\mathcal{W}$ is an UMFF if and only if whenever $\boldsymbol{xy}, \boldsymbol{yz} \in \mathcal{W}$ for some nonempty $\boldsymbol{y}$, then $\boldsymbol{xyz} \in \mathcal{W}$.*

Lemma 2.9. *The set of B-words forms an UMFF.*

Proof. Let $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ be three non-empty words on Σ such that $\boldsymbol{xy}$ and $\boldsymbol{yz}$ are B-words. We will use the fact that Lyndon words form an UMFF [CFL58,

DD03, Day11] and therefore satisfy Lemma 2.8. Since B-words must start with a 1, we have $\boldsymbol{xy}[1] = \boldsymbol{yz}[1] = 1$ and therefore the overlap $\boldsymbol{y}[1] = 1$. We proceed to consider four cases arising from Theorem 2.5, and we assume $\boldsymbol{x}$ and $\boldsymbol{z}$ are nonempty for otherwise it is trivial. We will also use the result that a word is a Lyndon word if and only if it is lexicographically less than any of its nonempty proper suffixes [Duv83, Smy03]; additionally, if $\boldsymbol{u}$, $\boldsymbol{v}$ are Lyndon words then $\boldsymbol{uv}$ or $\boldsymbol{vu}$ is a Lyndon word.

i. $\hat{N}(\boldsymbol{xy})$ and $\hat{N}(\boldsymbol{yz})$ form Lyndon words over $\{1 > 2 > 3 > \cdots\}$. Since Lyndon words form an UMFF, $\hat{N}(\boldsymbol{xyz})$ is a Lyndon word over $\{1 > 2 > 3 > \cdots\}$.

ii. $\hat{N}(\boldsymbol{xy})$ and $\hat{N}(\boldsymbol{yz})$ form repetitions of Lyndon words over $\{1 > 2 > 3 > \cdots\}$, and $\hat{Z}(\boldsymbol{xy})$ and $\hat{Z}(\boldsymbol{yz})$ form Lyndon words over $\{0 < 1 < 2 < 3 < \cdots\}$. Suppose that $\hat{N}(\boldsymbol{xy}) = \boldsymbol{n^p}$ and $\hat{N}(\boldsymbol{xyz}) = \boldsymbol{n^q}$, where $p < q$, then $\zeta(\boldsymbol{xyz})$ is a Lyndon word over the alphabet $\{1^{|\boldsymbol{n}|} < 1^{|\boldsymbol{n}|-1}2 < 1^{|\boldsymbol{n}|-1}3 < \cdots\}$. Otherwise, $\hat{N}(\boldsymbol{xyz})$ is not a repetition, and consider the seed $\boldsymbol{n}$ of $\hat{N}(\boldsymbol{xy})$ and the seed $\boldsymbol{m}$ of $\hat{N}(\boldsymbol{yz}) = \boldsymbol{m^r}$. Let $i > 1$ be the index such that $\hat{N}(\boldsymbol{y})[1] = \hat{N}(\boldsymbol{xy})[i]$; then there are two cases for the overlap $\hat{N}(\boldsymbol{y})$.

If i indexes the start of some $\boldsymbol{n}$ in $\boldsymbol{n^p}$ then, by the maximality of Lyndon words, $\boldsymbol{m}$ cannot be shorter than $\boldsymbol{n}$. If $\boldsymbol{m}$ is the same length as $\boldsymbol{n}$, then $\hat{N}(\boldsymbol{xyz})$ forms a repetition as above; otherwise the suffix of $\hat{N}(\boldsymbol{xy})$ starting at i is a proper prefix of $\boldsymbol{m}$ and $\hat{N}(\boldsymbol{xyz})$ is a Lyndon word over $\{1 > 2 > 3 > \cdots\}$.

If on the other hand a prefix $\hat{N}(\boldsymbol{v})$ of $\hat{N}(\boldsymbol{y})$ is a proper suffix of some $\boldsymbol{n}$, then, since $\hat{N}(\boldsymbol{n}) < \hat{N}(\boldsymbol{v})$, it must be the pth $\boldsymbol{n}$. Writing $\boldsymbol{n} = \boldsymbol{uv}$, and applying Lemma 2.8 with nonempty overlap $\hat{N}(\boldsymbol{v})$, we have $\hat{N}(\boldsymbol{um})$ is a Lyndon word over$\{1 > 2 > 3 > \cdots\}$ with prefix $\boldsymbol{n}$, and likewise $\hat{N}(\boldsymbol{n^{p-1}um})$ and $\hat{N}(\boldsymbol{n^{p-1}um^r})$. Therefore $\hat{N}(\boldsymbol{xyz})$ forms a Lyndon word as required.

iii. $\hat{N}(\boldsymbol{xy})$ satisfies Case [i] and $\hat{N}(\boldsymbol{yz})$ satisfies Case [ii].
Let $\hat{N}(\boldsymbol{y}) = \boldsymbol{m_1 m_2 \ldots m_s}[1 \ldots t]$, where $\boldsymbol{m_1} = \boldsymbol{m_2} = \cdots = \boldsymbol{m_s} = \boldsymbol{m}$ and $s < r$, $t \leq |\boldsymbol{m}|$ or $s \leq r$, $t < |\boldsymbol{m}|$. If $t = |\boldsymbol{m}|$, then since $\boldsymbol{m}$ is a suffix of $\hat{N}(\boldsymbol{xy})$ we have $\hat{N}(\boldsymbol{xy}) < \hat{N}(\boldsymbol{m})$, and it follows that $\hat{N}(\boldsymbol{xym^{r-s}})$ is a Lyndon word. If $t < |\boldsymbol{m}|$, then write $\boldsymbol{m} = \boldsymbol{uv}$ where $\boldsymbol{u} = \boldsymbol{m_s}[1 \ldots t]$ and $\boldsymbol{v} = \boldsymbol{m_s}[t+1 \ldots |\boldsymbol{m}|]$. Applying Lemma 2.8 with nonempty overlap $\hat{N}(\boldsymbol{u})$, we have that $\hat{N}(\boldsymbol{xyv})$ is a Lyndon word with suffix $\hat{N}(\boldsymbol{m})$. Furthermore, $\hat{N}(\boldsymbol{xyvm^{r-s}})$ is a Lyndon word. In either case $\hat{N}(\boldsymbol{xyz})$ forms a Lyndon word over $\{1 > 2 > 3 > \cdots\}$.

iv. $\hat{N}(\boldsymbol{xy})$ satisfies Case [ii] and $\hat{N}(\boldsymbol{yz})$ satisfies Case [i]. Consider the prefix of $\hat{N}(\boldsymbol{yz})$ which overlaps the ith repetition $\boldsymbol{n_i} = \boldsymbol{n}$ in $\hat{N}(\boldsymbol{xy})$ with minimal i, that is $\hat{N}(\boldsymbol{yz})[1 \ldots \hat{N}(\boldsymbol{xy})[i|\boldsymbol{n}|] = \hat{N}(\boldsymbol{xy})[d \ldots e] = \boldsymbol{v}$. So we can write $\boldsymbol{n_i} = \boldsymbol{uv}$, $\boldsymbol{v} \neq \varepsilon$.

If d indexes the start of the repetition $\boldsymbol{n_i}$, that is $d = (i-1)|\boldsymbol{n}| + 1$, then the Lyndon word $\hat{N}(\boldsymbol{yz})$ has prefix $\hat{N}(\boldsymbol{n_i})$ and so $\hat{N}(\boldsymbol{n^{i-1}yz})$ is also a Lyndon word.

Otherwise, d does not index the start of $\boldsymbol{n_i}$. Since $\hat{N}(\boldsymbol{n_i})$ and $\hat{N}(\boldsymbol{yz})$ are both Lyndon words with nonempty overlap $\hat{N}(\boldsymbol{v})$, then Lemma 2.8 applies, and $\hat{N}(\boldsymbol{u}yz)$ is a Lyndon word with prefix $\hat{N}(\boldsymbol{n_i})$. As above $\hat{N}(\boldsymbol{n^{i-1}uyz})$ is also a Lyndon word.

In either case $\hat{N}(\boldsymbol{xyz})$ forms a Lyndon word over $\{1 > 2 > 3 > \cdots\}$.

We conclude that the set of B-words satisfies the xyz Lemma and hence forms an UMFF. □

For further results, examples and discussion on B-words, see [DD96, DD03].

2.2. Combinatorics of Rotations in B-order

Analogously to the classic lexorder BWT [BW94], the method for a binary block order transform, B-BWT, is specified as: order the conjugates of an input string $\boldsymbol{x}$ of length n into B-order, giving the $n \times n$ B-BWT matrix M; the ***transform*** T is then defined to be the last right-hand column of the matrix M, that is $M[i, n]$ for $1 \leq i \leq n$ – in Example 2.7 the B-BWT transform is $T = 010100111$. The particular cases considered here are where $\boldsymbol{x}$ is assumed to be a B-word of either type; hence we work with Definition 2.1(ii) & (iii) (part (i) does not apply to cyclic rotations).

From Definition 2.1(ii), we see that the sizes of the left-most N_1 components of rows in M are decreasing. This partitions M into ***groups*** $G_t, G_{t-1}, \ldots, G_1$ of rows where group G_i is all rows where the first occurrence of a block of 1's is of size i, $N_1(\boldsymbol{x}) \geq i \geq 1$. Hence in Example 2.7, M has groups: G_2, rows 1 - 4; and G_1, rows 5 - 9.

We proceed to establish some new results for B-words, first for primitive then general and finally type repetition, which although fairly simple, help to unravel patterns in the maze of bits which lead to the transforms and inverses.

The next lemma shows that rotations of the Z suffixes of B-prim words which end with 0 can just be inserted directly into the matrix M.

Lemma 2.10. *Let the binary string $\boldsymbol{r}$ be a row in the B-BWT matrix M of a B-prim word with $\hat{\boldsymbol{r}} = Z_0N_1Z_1N_2Z_2 \ldots N_rZ_r$ and $0 < Z_r = k$. Then $\boldsymbol{r} \ll (\boldsymbol{r}^-)^1 \ll \cdots \ll (\boldsymbol{r}^-)^k$ in M; furthermore, these rotations are consecutive in M.*

Proof. Definition 2.1(ii) shows that rows in M with the same N values will be consecutive in M. Rotating a 0 suffix bit leaves the N values of a string unchanged but increments the Z_0 component - hence Definition 2.1(iii) applies. □

Corollary 2.11. *Let $\boldsymbol{r}$ be as in Lemma 2.10. Then the sequence of rotations $\boldsymbol{r}, (\boldsymbol{r}^-)^1, \ldots, (\boldsymbol{r}^-)^k$ corresponds to a substring of the form 0^k in the transform T.*

That is, each Z block in a given B-prim word $\boldsymbol{x}$ will correspond to a Z suffix of some row in M, which by the subsequent rotations, will give a run of 0^k in the transform T – a key insight for computing the inverse of the transform.

The N blocks of unit size, which form group G_1, also have particular importance for initiating the inverse transforms; the next corollary addresses the formation of this group.

Corollary 2.12. *Following the rotation of a complete Z block suffix in (either type) M, rotating the next adjacent 1 bit gives a row in group G_1 of M.*

Hence rotating the right-most bit of any N block (of any size) in the input string $\boldsymbol{x}$ generates a rotation in group G_1. We also observe that in any row of the matrix M, at most one block of either type N or Z can be split, thus giving a bordered row in $\mathcal{R}_1^b$ or $\mathcal{R}_0^b$ respectively.

The following lemma will be applied later to show that, given a partial B-BWT matrix of a B-prim word derived from a suffix array, the full B-BWT matrix can be readily constructed.

Lemma 2.13. *Let $\boldsymbol{r_i}$, $\boldsymbol{r_j}$ be two rows in a B-BWT matrix M with $\boldsymbol{r_i} \ll \boldsymbol{r_j}$. Suppose that $\boldsymbol{r_i}[n] = \boldsymbol{r_j}[n]$, and also that these rows do not satisfy $\boldsymbol{r_i}[1] = 0$ and $\boldsymbol{r_j}[1] = 1$. Then ${\boldsymbol{r_i}}^- \ll {\boldsymbol{r_j}}^-$.*

Proof. Since the strings $\boldsymbol{r_i}$, $\boldsymbol{r_j}$ are conjugates, they cannot differ only on their right-hand blocks of bits, and so the ordering of $\boldsymbol{r_i}$ and $\boldsymbol{r_j}$ is determined prior to the last blocks. Consider the cases for the first bits:

- $\boldsymbol{r_i}[1] = \boldsymbol{r_j}[1]$. Then clearly Definition 2.1(ii) & (iii) apply to each of the eight cases (start 0 or 1; end 0 or 1; differ on 0's or 1's) as appropriate, giving ${\boldsymbol{r_i}}^- \ll {\boldsymbol{r_j}}^-$.
- $\boldsymbol{r_i}[1] = 1$ and $\boldsymbol{r_j}[1] = 0$. If $\boldsymbol{r_i}[n] = \boldsymbol{r_j}[n] = 1$, then $N_1((\boldsymbol{r_i})_1^-) > N_1((\boldsymbol{r_j})_1^-) = 1$, and so by Definition 2.1(ii), ${\boldsymbol{r_i}}^- \ll {\boldsymbol{r_j}}^-$. Otherwise $\boldsymbol{r_i}[n] = \boldsymbol{r_j}[n] = 0$, whereby $1 = Z_0((\boldsymbol{r_i})_0^-) < Z_0((\boldsymbol{r_j})_0^-)$, and so either the order of ${\boldsymbol{r_i}}^-$ and ${\boldsymbol{r_j}}^-$ is determined by the N components which haven't changed, or the order is determined by Definition 2.1(iii), and in either case ${\boldsymbol{r_i}}^- \ll {\boldsymbol{r_j}}^-$.

□

In other words, if two strings in the matrix both end with either 0 or 1, but do not start with 0 and 1 respectively, then the B-order of the strings is maintained when rotating those last bits - note that this does not apply to the case of different end bits. In particular, this lemma leads the way to applying existing knowledge on the order of strings to the ordering of their rotations.

The Last First mapping for the classic Burrows Wheeler transform utilizes the fact that the relative positions of letters correspond in the first and last columns in the BWT matrix – hence it is applied for recovering a string from the computed transform. Lemma 2.13 shows that this correspondence is not

the case with binary bits: see Example 2.7 where the 1st, 2nd, 3rd, 4th and 5th 1 bits in the Last column map to the 3rd, 4th, 5th, 1st and 2nd 1 bits in the First column respectively. Hence we distinguish between matching suffix block pairs of the form 10^i with $i > 0$ and with $i = 0$ in the mapping in Section 3.4.

Lemma 2.14. *Suppose that the binary strings* $\boldsymbol{u} = 1 \dots 10^i$ *and* $\boldsymbol{v} = 1 \dots 10^j$ *where* $\boldsymbol{u} \ll \boldsymbol{v}$ *are conjugates of one B-prim word. If* $i, j > 0$ *or* $i, j = 0$*, then* $(\boldsymbol{u}^-)^{i+1} \ll (\boldsymbol{v}^-)^{j+1}$.

Proof. Consider the two cases.

Case $i, j > 0$**.** Let $\hat{\boldsymbol{u}} = N_1 Z_1 N_2 Z_2 \dots N_r Z_r$ and $\hat{\boldsymbol{v}} = N'_1 Z'_1 N'_2 Z'_2 \dots N'_s Z'_s$. Since $\boldsymbol{u}$, $\boldsymbol{v}$ are unbordered conjugates of a Lyndon word, then Definition 2.1(iii) cannot apply; furthermore, $r = s$. Then $\hat{N}(\boldsymbol{u}) \neq \hat{N}(\boldsymbol{v})$ and by Definition 2.1(ii) some $N_t > N'_t$. Further, since $\boldsymbol{u}$, $\boldsymbol{v}$ are conjugates with the same weight, they cannot differ only on the last N_r, N'_s pair. So assume $t < r$, and then the order of $\boldsymbol{u}$ or $\boldsymbol{v}$ is not changed by the rotations of the respective suffixes 10^i and 10^j. In this case, rotations of suffixes always map to group G_1 - taking you down the matrix.

Case $i, j = 0$**.** Here $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{R}_1^b$, and let $\hat{\boldsymbol{u}} = N_1 Z_1 N_2 Z_2 \dots N_{r+1}$ and $\hat{\boldsymbol{v}} = N'_1 Z'_1 N'_2 Z'_2 \dots N'_{s+1}$; then $r+1 = s+1$. Suppose Definition 2.1(iii) applies, then $N_1 + N_{r+1} = N'_1 + N'_{s+1}$ and $N_q = N'_q$ for $2 \leq q \leq r$, contradicting the primitive property of the B-prim word. Therefore Definition 2.1(ii) applies as above. In this case, the rotation of a suffix bit always increments the group size of each conjugate - taking you up the matrix, possibly to different groups.

□

Since Lemma 2.14 holds for all such $\boldsymbol{u}$, $\boldsymbol{v}$, then we see that the kth occurrence of a suffix 10^i in Last maps to the kth occurrence of 10^i as a prefix in group G_1, and the kth occurrence of a row belonging to $\mathcal{R}_1^b$ in G_h maps to the kth occurrence of the prefix 1^{h+1} in group G_{h+1}.

We conclude these observations on rotations with a result for the repetition case of Theorem 2.5, which shows that rotations of B-rep words with the same $\hat{N}$ encodings, arising from repetitions, are adjacent in the B-BWT matrix M.

Lemma 2.15. *Suppose that the rotations* $\boldsymbol{r_i}, \boldsymbol{r_{i+1}}, \dots, \boldsymbol{r_{i+j}}$ *of a B-rep word* $\boldsymbol{x}$ *belong to the set* $\mathcal{R}_1^u$ *and satisfy* $\hat{N}(\boldsymbol{r_i}) = \hat{N}(\boldsymbol{r_{i+1}}) = \dots = \hat{N}(\boldsymbol{r_{i+j}})$*. Then these* $j+1$ *rows are consecutive in the B-BWT matrix M; furthermore, the rotations of their Z suffixes all immediately follow* $\boldsymbol{r_i}, \boldsymbol{r_{i+1}}, \dots, \boldsymbol{r_{i+j}}$ *in the matrix M.*

Proof. Suppose that the conjugate $\boldsymbol{q}$ of $\boldsymbol{x}$ has identical N components to $\boldsymbol{r_i}$, that is $\hat{N}(\boldsymbol{q}) = \hat{N}(\boldsymbol{r_i})$, and also that $\boldsymbol{q}$ has a non-empty Z prefix, so $Z_0(\boldsymbol{q}) > 0$. Since the rotations $\boldsymbol{r_i}, \dots, \boldsymbol{r_{i+j}}$, which are generated by repetitions, all have empty Z prefixes ($Z_0 = 0$), then, by Definition 2.1(iii), these $j + 1$ rows must

all precede $\boldsymbol{q}$ in M – the next conjugate adjacent to $\boldsymbol{r_{i+j}}$ in M must have the same $\hat{N}$ encoding and $Z_0 = 1$, followed by those with $Z_0 \geq 1$ prefixes but all with the same $\hat{N}$ encoding. Suppose now that $\boldsymbol{q}$ has distinct N components to $\boldsymbol{r_i}$, that is $\hat{N}(\boldsymbol{q}) \neq \hat{N}(\boldsymbol{r_i})$. Then by Definition 2.1(ii), $\boldsymbol{q}$ either precedes $\boldsymbol{r_i}$, or comes after $\boldsymbol{r_i}, \ldots, \boldsymbol{r_{i+j}}$ and all the rotations of their Z suffixes. □

We are now ready to introduce the binary block order Burrows-Wheeler type transforms. For B-prim words, the transform is presented in Section 3, followed by the computation of its inverse in Section 3.1; for B-rep words, the transform is given in Section 4 and the inverse in Section 4.1.

3. The *B*-BWT Binary Transform - Primitive Case

The key computation in the B-BWT transform is the B-ordering of all cyclic rotations of an input $\boldsymbol{x}$ to obtain the B-BWT matrix M – the transform T is then readily extracted from the right-hand column of M. For the B-sorting we apply a linear time and space suffix array construction for sorting string suffixes and hence conjugates – we will basically apply such a method as a "black box"; for details the interested reader is referred to Kärkkäinen and Sanders [KS03], Ko and Aluru [KA03], or Nong, Zhang and Chan [NZC09].

These suffix array techniques apply lexorder to compare string suffixes which can be readily adapted to binary B-order comparison – Claim 2.2 shows that the linear complexity is maintained. Note that for the modification for the B-order suffix array, the suffixes relate to conjugates which all have the same weight ω, therefore part (i) of Definition 2.1 can be ignored and parts (ii) & (iii) adapted and applied as follows. Suppose that $\boldsymbol{u}$ and $\boldsymbol{v}$ are nonempty suffixes such that $|\boldsymbol{u}| < |\boldsymbol{v}|$. If some $\hat{N}_i(\boldsymbol{u}) \neq \hat{N}_i(\boldsymbol{v})$, then the order is determined by the two N_i values according to Definition 2.1(ii). If $\hat{N}(\boldsymbol{u}) = \hat{N}(\boldsymbol{v}) \neq \varepsilon$, then $\hat{Z}_0(\boldsymbol{u}) \neq \hat{Z}_0(\boldsymbol{v})$, and the order is determined according to Definition 2.1(iii). If $\hat{N}(\boldsymbol{u})$ is empty, then $\boldsymbol{u}$ is a single Z block, and so we append $1^{N_1(\boldsymbol{x})}$ as a suffix to $\boldsymbol{u}$; similarly for $\hat{N}(\boldsymbol{v})$. For cases when $\hat{N}(\boldsymbol{u})$ is a proper prefix of $\hat{N}(\boldsymbol{v})$ we apply:

Lemma 3.1. *Let $\boldsymbol{b}$ and $\boldsymbol{bd}$ be distinct nonempty proper suffixes of a B-prim word $\boldsymbol{x}$, where $\boldsymbol{b}$ starts with a suffix of the j^{th} N block corresponding to $\hat{N}_j(\boldsymbol{x})$, and $\boldsymbol{bd}$ starts with a suffix of the i^{th} N block corresponding to $\hat{N}_i(\boldsymbol{x})$, $i < j$. With $\boldsymbol{x} = \boldsymbol{ab} = \boldsymbol{cbd}$, where $\boldsymbol{a, b, c, d} \neq \varepsilon$, then the conjugates $\boldsymbol{ba}$ and $\boldsymbol{bdc}$ of $\boldsymbol{x}$ satisfy $\boldsymbol{ba} \ll \boldsymbol{bdc}$.*

Proof. Note that since $\hat{N}(\boldsymbol{x})$ is a Lyndon word, and hence border-free, then $\hat{N}(\boldsymbol{b})$ cannot also be a prefix of $\hat{N}(\boldsymbol{x})$. Further, $|\hat{N}_1(\boldsymbol{bd})| \leq |\hat{N}_i(\boldsymbol{x})|$ and $|\hat{N}_1(\boldsymbol{b})| \leq |\hat{N}_j(\boldsymbol{x})|$.

Consider B-ordering the conjugates $\boldsymbol{ba}$ and $\boldsymbol{bdc}$ by first applying lexorder (with $>$) to $\hat{N}(\boldsymbol{ba})$ and $\hat{N}(\boldsymbol{bdc})$. Since the prefixes of the conjugates are both $\boldsymbol{b}$, and $\boldsymbol{c} \neq \varepsilon$, then $|\boldsymbol{a}| = |\boldsymbol{dc}|$ and $|\boldsymbol{a}| > |\boldsymbol{d}|$. From the Lyndon properties of $\hat{N}(\boldsymbol{x})$ we have $\hat{N}(\boldsymbol{x}) \ll \hat{N}(\boldsymbol{d})$, but since $\hat{N}(\boldsymbol{x})$ is border-free $\hat{N}(\boldsymbol{d})$ is not a prefix of $\hat{N}(\boldsymbol{a})$. Hence, by minimality of $\boldsymbol{x}$, $\hat{N}(\boldsymbol{a})[h] > \hat{N}(\boldsymbol{d})[h]$ for some $1 \leq h \leq |\boldsymbol{d}|$,

and therefore the order is not changed by appending $\boldsymbol{c}$ to $\boldsymbol{d}$; we conclude that $\boldsymbol{ba} \ll \boldsymbol{bdc}$. □

Hence, although suffixes relating to conjugates may not have the same length, we can order them similarly to lexorder by using their integer $\hat{N}$ and $\hat{Z}$ values – intermediary values relating to split blocks can be recorded during a right-to-left Run Length Encoding scan. Depending on the implementation, the B-order suffix array can be constructed just for the N values, and then the Z prefixes and suffixes entered into M according to Lemma 2.10. Overall, by modifying the ordering for a classic suffix array technique we get:

Claim 3.2. *The B-BWT matrix M can be computed in time linear in the bit length, n, of the input $\boldsymbol{x}$.*

Finally, the transform T is trivially extracted from M as $M[i, n],\ 1 \leq i \leq n$.

Example 3.3. *Consider the B-prim word $\boldsymbol{x} = 1110011001110101100$ where $\hat{\boldsymbol{x}} = 03222311122$. Then the $n \times n$ matrix of B-sorted rotations is given by M_x^B:*

1	1	1	0	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0
0	1	1	1	0	0	1	1	0	0	1	1	1	0	1	0	1	1	0
0	0	1	1	1	0	0	1	1	0	0	1	1	1	0	1	0	1	1
1	1	1	0	1	0	1	1	0	0	1	1	1	0	0	1	1	0	0
0	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0	1	1	0
0	0	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0	1	1
1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	1	0	1	0
0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	1	0	1
1	1	0	0	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
0	1	1	0	0	1	1	1	0	1	0	1	1	0	0	1	1	1	0
0	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0	1	1	1
1	1	0	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0	1
1	1	0	1	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1
1	0	0	1	1	1	0	0	1	1	0	0	1	1	1	0	1	0	1
1	0	0	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0	1
1	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	1	0
0	1	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1	1
1	0	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0	1	1
1	0	1	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	1

The B-BWT transform is $T = 0^2 10^2 1010^2 1^5 01^3$.

The following small example shows better clustering of bits with the B-BWT compared to the classic transform with binary lexorder.

Example 3.4. *Consider the B-prim word $\boldsymbol{x} = 110100010$. Then the B-BWT is $010^3 101^2$ while the binary lexorder transform, with $0 < 1$, is 10101010^2 (hence $0^2 1010101$ with $1 < 0$).*

3.1. The Inverse B-BWT - Primitive Case

In this section we establish that the B-BWT is bijective for B-prim words by showing how to recover the B-word input losslessly from the transform.

We will assume that the given encoded B-prim word $\hat{\boldsymbol{x}} = N_1 Z_1 N_2 Z_2 \dots N_r Z_r$ has $r \geq 2$, for otherwise the inverse is trivial:

Lemma 3.5. *The encoding of a B-prim word $\boldsymbol{x}$ is $\hat{\boldsymbol{x}} = N_1 Z_1$ if and only if the last column T of the associated B-BWT matrix M has the form $0^{Z_1} 1^{N_1}$.*

Proof. Given a B-prim word $\boldsymbol{x}$ such that $\hat{\boldsymbol{x}} = N_1 Z_1$, then applying Definition 2.1: the first Z_1 rows in M end with 0 due to rotations $\boldsymbol{r}$ of $\boldsymbol{x}$ of the form $\boldsymbol{r}_0^-$, and the ordering of Zs using $<$; the following N_1 rows end with 1 due to rotations of the form $\boldsymbol{r}_1^-$, and the ordering of Ns using $>$.

Given a transform $T = 0^{Z_1} 1^{N_1}$, then by Corollary 2.11 there is only one Z block in $\boldsymbol{x}$ of length Z_1. Since the B-word starts 1 and ends 0, then it must be $1^{N_1} 0^{Z_1}$, and hence type B-prim. □

3.2. Outline of the Inverse.

An outline of the steps for the B-BWT inversion algorithm is as follows:

- Assume that the input $\boldsymbol{x}$ to be recovered is a B-prim word, and hence in the set $\mathcal{R}_1^u$.
- The input for the inversion is the binary transform T: the last column of bits of the B-BWT matrix M.
- Determine the number of block pairs in $\boldsymbol{x}$ from T – Algorithm 2.
- Find the start index of each group in M – Algorithm 3.
- Obtain the indexes for the changes in the values of the second N blocks, N_2, in M (since $\hat{N}(\boldsymbol{x})$ is primitive the N_2 values cannot all be the same) – Algorithm 4.
- Apply the N_2 indexes to perform a Last First matching by using the given transform T and block pairs to recover the input string – Algorithm 5.

3.3. Indexing for the Inverse

We now present the details of the inversion algorithm, separated into simple procedures – hence we assume the variables are global as appropriate.

In order to index the matrix M, the transform is first scanned to find the number of block pairs in the input string $\boldsymbol{x}$: applying Corollary 2.11, the number of runs of 0's in the transform gives the value of r in $\hat{\boldsymbol{x}}$, that is the number of block pairs in $\boldsymbol{x}$.

Using the linear-time procedure Compute-BP, we next find the start index of each group in M: due to the splitting of N blocks there must be a group for

Algorithm 2 Compute the number of block pairs in the input $\boldsymbol{x}$ from T.

```
procedure COMPUTE-BP(T, n)
    ▷ scan T and calculate the number r of block pairs in x

    i := 1;                          ▷ initialize the transform index
    blockpairs := 0;                 ▷ initialize the number of block pairs
    while i ≤ n do
        while T[i] = 0 do
            i := i + 1;
        end while
        blockpairs := blockpairs + 1;        ▷ apply Corollary 2.11
        while i ≤ n and T[i] = 1 do
            i := i + 1;
        end while
    end while

    return blockpairs
end procedure
```

each size $1, \ldots, N_1$. From $r = blockpairs$, we obtain the number, t, of rows in group G_1 as follows. For each unbordered row ending $N_i > 1$, by rotating the suffix bit of the block 1^{N_i}, the block becomes a split block, and the rotation will have the form $10 \cdots 1^{N_i - 1}$ and hence is in $\mathcal{R}_1^b$ and G_1. For any $N_i = 1$, the block will occur in $\boldsymbol{x}$ as a substring of the form $\cdots 0^{j_i} 10^{k_i} \cdots$, then by Corollary 2.11, it is associated with the substring $0^{k_i}1$ in the transform T; the last row related to this substring in T has the form $0^{k_i} \cdots 0^{j_i}1$. Hence $t = r + j_i$ for all such i. Note that we can also fill in the prefixes of 0's for these rows in M.

Similarly, a rotation of each of the s bordered rows in G_1 generates a row in G_2, and so we also have the number of blocks N_i where $N_i = 2$; accordingly, there are $s + l_i$ rows in group G_2, where l_i is the number of rotations associated with each N_i block occurring in $\boldsymbol{x}$ as $\cdots 0^{l_i} 110^{m_i} \cdots$ – and so the bootstrapping continues until the first left-hand column of Ns is completed (including Z prefixes where $Z > 0$). To summarize: after deducing the number of rows in group G_1 from the number of block pairs, the size of the next larger N value group is obtained iteratively, moderated by the Z blocks, since their rotations do not form rows in the neighbouring group – see Algorithm 3.

Observing that the algorithm performs a single pass through the transform, with associated constant work for each bit in the transform, we get:

Claim 3.6. *Procedure* COMPUTE-GI(T, n) *runs in time linear in the bit length of the transform.*

In order to perform a Last First type mapping in Algorithm 5, we will need information on the second column of 1's in M, namely the index of each change

Algorithm 3 Compute the index of each group in M from T.

```
procedure COMPUTE-GI(T, n)
    G[1...];                ▷ declare an array list for group indexes in M
    j := 1;                 ▷ initialize the group index
    G[j] := n;              ▷ initialize the index of the start of group G_1
    nonsplit := 0;          ▷ initialize number of non split blocks in each group
    exp := 0;               ▷ initialize the exponent for a Z block
    bool := false;          ▷ set flag for non split blocks
                            ▷ blockpairs is computed in Algorithm 2

    while G[j] > 1 do       ▷ find the indexes for group G_1 up to group G_{N_1}
        while G[j] > n − blockpairs − exp do
            while G[j] > n − blockpairs − exp and T[G[j]] = 1 do ▷ N suffix
                G[j] := G[j] − 1
            end while
            while G[j] > n−blockpairs−exp−1 and T[G[j]] = 0 do ▷ Z suffix
                G[j] := G[j] − 1;
                exp := exp + 1;
                bool := true
            end while
            if bool then nonsplit := nonsplit + 1; bool := false
            end if
        end while

        n := G[j];          ▷ iterate through the groups
        G[j] := G[j] + 1; j := j + 1;
        G[j] := n;          ▷ initialize the index of the start of next group
        blockpairs := blockpairs−nonsplit;  ▷ adjust for G_1 (Corollary 2.12)
        nonsplit := 0;
        exp := 0
    end while

    max := j − 1;           ▷ max = N_1 is used in Algorithms 4 and 5
    return G[1...]
end procedure
```

in N_2 value. The B-BWT matrix consists of groups G_i, with $i > 1$, and group G_1 – we use this partition to deduce the N_2 column.

Procedure COMPUTE-INDEXN$_2$ first scans down the transform from G_{max} to G_2 detecting Z suffixes - rotations of these suffixes generate rows in G_1 by Corollary 2.12; the relevant point is that for a Z block suffix of a row in group G_i of the form $1^i \dots 10^j$, then $j + 1$ rotations yield a row in G_1 with prefix $10^j1^i0\dots$, and so we know the N_2 value i as it relates to a non-split block. This process gives all entries in the array $IndexN_2$ for G_1. Using the partially completed array $IndexN_2$, the algorithm then sweeps back up the transform incrementally deducing the entries for larger groups. Note that although we compute a $max \times max$ array, the number of entries is less than n, hence an $O(n)$ method.

3.4. The Last First Mapping

We finally emulate the Last First mapping of the classic BWT in which the ith occurrence of a letter in the last column is mapped to the ith occurrence of that letter in the first column; however, this property only holds for 0 bits in the B-BWT matrix M (Lemma 2.10, Corollary 2.12, Lemma 2.13). Therefore, using Lemma 2.14, we apply the property to a partition of M: unbordered and bordered rows, both beginning with 1, which map to group G_1 and groups G_i, $i > 1$, respectively. The B-BWT Last First mapping thus rotates any suffixes of 0's until a 1 bit is reached, and then the prefix matching process is resumed, where a prefix always starts with a 1 bit.

From the first run 0^j in the transform T and Corollary 2.11, we know that the input B-word $\boldsymbol{x}$ ends 10^j. From Algorithm 3 we know the value $N_1 = max$ and so, since we assume $r \geq 2$, the string formed by $(\boldsymbol{x}^-)^{j+1}$ has the prefix $10^j1^{N_1}0$, which, by Definition 2.1, must be the first row, rotation $\boldsymbol{r_1}$ say, in group G_1. This forms the start of the Last First pattern matching process, which proceeds as follows. Consider $T[i]$ corresponding to $\boldsymbol{r_1}$: suppose $T[i] = 1$, then by Definition 2.1, $(\boldsymbol{r_1})^-$ is the first row in group G_2 as it begins with the prefix $110^j1^{N_1}0$. Otherwise, $T[i] = 0$, and we perform $(\boldsymbol{r_i})^-$ repeatedly, that is scan T, until we find a 1 bit, and hence by Corollary 2.12 the final rotation is in group G_1. The method oscillates through the groups until n bits have been recovered, thus forming a cycle.

In order to implement the Last First mapping, we need the index of the next row to obtain the prefix of the string being incrementally developed. If the next row is in group G_i, then it will come after all rows in any groups with larger i value. Furthermore, the next row will also come after any rows in G_i whose prefixes have already been selected and concatenated to the evolving string. For this we track the indexes of both N_1, with the variable $currG$, and N_2, with $prevG$, for rows as we trace the cycle of rotations to reconstruct the input $\boldsymbol{x}$.

In the implementation in Algorithm 5 we only visit each bit in the transform once, with constant work for each step, hence:

Claim 3.7. *Procedure* COMPUTE-LF(T, n) *can be implemented in linear time.*

Algorithm 4 Compute indexes for the second column of 1's, N_2, in M from T.

```
procedure COMPUTE-INDEXN2(T, n)
    IndexN2[max, max];      ▷ declare an array for N2 indexes of each group
    ▷ find N2 indexes for group G1
    runs := 0;              ▷ runs is the number of runs of 0's in T for a group
    IndexN2[1, max] := G[1];    ▷ max and G are computed in Algorithm 3
    for i = max − 1 downto 1 do     ▷ i indexes groups
        j = G[i + 1]; bool := false;    ▷ j indexes rows in the current group
        repeat
            if T[j] = 0 then bool := true
            else if bool then runs := runs + 1; bool := false
            end if
            j := j + 1
        until j = G[i]
        h := IndexN2[1, i + 1]; N := 0;         ▷ h indexes rows in group G1
        repeat
            if T[h] = 1 then N := N + 1
            end if      ▷ N counts N suffixes
            h := h + 1
        until N = runs
        IndexN2[1, i] := h; runs := 0
    end for     ▷ iterate from G1 to Gmax with each group bootstrapping the next
    IndexN2[1, 0] := n + 1;         ▷ set a dummy entry for the loop
    last := G[1] − 1;               ▷ index of last row of current group
    for i = 1 to max − 1 do
        for j = 1 to max do
            if last > 0 then
                N := 0; bool := false;      ▷ k indexes rows in previous group
                for k = IndexN2[i, j] to IndexN2[i, j − 1] − 1 do
                    if T[k] = 0 then bool := true
                    else if bool then bool := false
                    elseN := N + 1
                    end if
                end for
                Ng := 0;            ▷ Ng counts N suffixes in current group
                while Ng < N do
                    if T[last] = 1 then Ng := Ng + 1; last := last − 1
                    end if
                    while T[last] = 0 do last := last − 1
                    end while
                end while
                IndexN2[i + 1, j] := last + 1
            end if
        end for
        IndexN2[i + 1, 0] := IndexN2[i, max];
    end for
    return IndexN2[max, max]
end procedure
```

Algorithm 5 Compute the Last First Mapping for a transform T.

procedure COMPUTE-LF(T, n)

$p := 10^{Z_r}$; ▷ initialize the pattern for the matching process
▷ from scanning the first run of 0's in T
$exp = 0$; ▷ initialize the exponent for a Z block
$currG := 1$; ▷ size of current group
$prevG := max$; ▷ size of previous group from Algorithm 3

while $|p| < n$ **do** ▷ iterate until n characters have been recovered

$t := IndexN_2[currG, prevG]$; ▷ obtain index of current row in M from
▷ $IndexN_2$ computed in Algorithm 4

if $T[t] = 1$ **then** ▷ the last bit of the current rotation in M is 1
$p := 1 \circ p$; ▷ concatenate the suffix bit 1 to the start of p
$IndexN_2[currG, prevG] := IndexN_2[currG, prevG] + 1$;
▷ increment the N_2 group row index
$currG := currG + 1$ ▷ increment the group size

else ▷ the last bit of the current rotation in M is 0
repeat $exp := exp + 1$; $t := t + 1$
until $T[t] = 1$
$p := 10^{exp} \circ p$; ▷ concatenate a Z block to the start of p
$IndexN_2[currG, prevG] := IndexN_2[currG, prevG] + exp + 1$;
▷ increment the N_2 group row index
$prevG := currG$;
$currG := 1$; ▷ the prefix 10^{exp} indicates a rotation in group G_1 (Corollary 2.12)
$exp := 0$ ▷ reset the exponent for a Z block

end if

end while

return $LF(\boldsymbol{x})$ ▷ the recovered input $\boldsymbol{x}$ is given by the pattern p
end procedure

At each step of the inversion, a block pair 10^i, $i \geq 0$, is concatenated to the evolving string and matched to the current least unselected prefix in M, hence:

Claim 3.8. *Procedure* COMPUTE-LF(T, n) *correctly recovers the input.*

4. The *B*-BWT Binary Transform - Repetition Case

With reference to Theorem 2.5, we start with an example of a B-rep word $\boldsymbol{x}$: the encoding $\hat{N}(\boldsymbol{x})$ forms a repetition of Lyndon words over $\{1 > 2 > \cdots\}$ while $\zeta(\boldsymbol{x})$ is a Lyndon word over the alphabet $\{1^p < 1^{p-1}2 < 1^{p-1}3 < \cdots\}$ (where p is the length of the repeated Lyndon word in $\hat{N}(\boldsymbol{x})$).

Example 4.1. *Consider the B-rep word* $\boldsymbol{x} = 11101100111011000$ *where* $\hat{\boldsymbol{x}} = 031223123$, $\hat{N}(\boldsymbol{x}) = (32)^2$ *with exponent of repetition 2, and* $\hat{Z}(\boldsymbol{x}) = 01213$. *Then the* $n \times n$ *matrix of B-sorted rotations is given by* M_x^B*:*

1	1	1	0	1	1	0	0	1	1	1	0	1	1	0	0	0
1	1	1	0	1	1	0	0	0	1	1	1	0	1	1	0	0
0	1	1	1	0	1	1	0	0	1	1	1	0	1	1	0	0
0	1	1	1	0	1	1	0	0	0	1	1	1	0	1	1	0
0	0	1	1	1	0	1	1	0	0	1	1	1	0	1	1	0
0	0	1	1	1	0	1	1	0	0	0	1	1	1	0	1	1
0	0	0	1	1	1	0	1	1	0	0	1	1	1	0	1	1
1	1	0	0	1	1	1	0	1	1	0	0	0	1	1	1	0
1	1	0	0	0	1	1	1	0	1	1	0	0	1	1	1	0
0	1	1	0	0	1	1	1	0	1	1	0	0	0	1	1	1
0	1	1	0	0	0	1	1	1	0	1	1	0	0	1	1	1
1	1	0	1	1	0	0	1	1	1	0	1	1	0	0	0	1
1	1	0	1	1	0	0	0	1	1	1	0	1	1	0	0	1
1	0	0	1	1	1	0	1	1	0	0	0	1	1	1	0	1
1	0	0	0	1	1	1	0	1	1	0	0	1	1	1	0	1
1	0	1	1	0	0	1	1	1	0	1	1	0	0	0	1	1
1	0	1	1	0	0	0	1	1	1	0	1	1	0	0	1	1

The B-BWT$(\boldsymbol{x}) = 0^5 1^2 0^2 1^8$ *with better clustering than the binary lexicographic* $BWT(\boldsymbol{x}) = 0^2 1^2 0^2 1^4 0^2 1^2 0^1 1^2$ *(with* $1 < 0$*).*

Example 4.2.
[i] Consider the B-rep word $\boldsymbol{x} = 11101101110011000$. *Then the B-BWT*$(\boldsymbol{x}) = 0^3 1^1 0^1 1^1 0^2 1^1 0^1 1^7$ *with one higher exponent than the binary lexicographic* $BWT(\boldsymbol{x}) = 0^3 1^2 0^1 1^4 0^1 1^2 0^2 1^2$ *(with* $1 < 0$*).*
[ii] Consider the B-rep word $\boldsymbol{x} = 11101011100100$. *Then the B-BWT*$(\boldsymbol{x}) = 0^3 1^4 0^2 1^1 0^1 1^3$ *with better clustering than the binary lexicographic* $BWT(\boldsymbol{x}) = 0^2 1^2 0^1 1^1 0^1 1^1 0^1 1^2 0^1 2^2$ *(with* $1 < 0$*).*

In general, for a B-rep word with repetition exponent k and s distinct N components, the B-BWT matrix M partitions as follows ($Z_{l,r}$ denotes the Z_r

suffix of row l):

k rows in the set $\mathcal{R}_1^u$ of the form $1^{max}\cdots 0$ [rows 1-2 above]

$h+k$ rows (possibly intermingled):
h rows in the set $\mathcal{R}_0^b$, where $h = \left(\sum_{l=1}^{k} Z_{l,r}\right) - k$ [rows 3-5 above]
k rows in the set $\mathcal{R}_0^u$ of the form $0\cdots 1$ [rows 6-7 above]

This structure is repeated for each group G_i with smaller N value, with the addition of a part consisting of bordered rows arising from splitting blocks of 1's with larger N values:

j rows in group G_i in the set $\mathcal{R}_1^b$,
where $j = (max - i)k$ [rows 12-13 above for group $G_i = G_2$]

With reference to Lemma 2.15, we will call a sub-matrix of M which consists of k rows in $\mathcal{R}_1^u$, having identical N components, along with their consecutive Z rotations, an N**-rep group** – there are two N-rep groups in Example 4.1: rows 1-7 and 8-11. The repetition of N blocks gives the k rows in $\mathcal{R}_1^u$ at the start of an N-rep group which yield a run of 0^k in the transform T. Note, however, that rotating Z suffixes doesn't necessarily give a run of 0's in the transform T as in Section 2.2, hence we cannot easily determine the number of block pairs here.

We proceed to briefly outline the linear B-sorting technique for B-rep words which gives the transform, followed by a description of the inversion method.

The B-sorting of all conjugates of a B-rep word is achieved in a similar way to B-prim words in Section 3, namely extending the definition of B-order and recording intermediate values during Run Length Encoding. So suppose again that $\boldsymbol{u}$ and $\boldsymbol{v}$ are suffixes of an input $\boldsymbol{x}$ such that $|\boldsymbol{u}| < |\boldsymbol{v}|$. Then the cases $\hat{N}_i(\boldsymbol{u}) \neq \hat{N}_i(\boldsymbol{v})$, $\hat{N}(\boldsymbol{u}) = \hat{N}(\boldsymbol{v}) \neq \varepsilon$, and empty $\hat{N}(\boldsymbol{u})$ or $\hat{N}(\boldsymbol{v})$ all follow as before. However, if $\hat{N}(\boldsymbol{u})$ is a proper prefix of $\hat{N}(\boldsymbol{v})$, then due to the repetitions of the N components, we need to order the suffixes using both the N and Z values to determine the least – to do this, append the first Lyndon seed of the $\hat{N}(\boldsymbol{x})$ repetition together with the adjacent $\hat{Z}(\boldsymbol{x})$ values to $\boldsymbol{u}$.

4.1. The Inverse B-BWT - Repetition Case

Rather naturally, the case of B-rep words with fixed repetition exponent k, yields the same basic Last First mapping as for B-prim words but adjusted according to Lemma 2.15.

There are three main steps to the inversion procedure:

1. Compute the array $IndexRepN_2$ of indexes of the changes in N_2 values – we omit an algorithm for computing this array since it can be achieved

similarly to Algorithm 4 as follows. Scan the transform and count the number of N-rep groups: they each start with a 0 and end after k 1's. During the scan also count in multiples of k the other rows, not belonging to N-rep groups, which are bordered and end with 1. Each N-rep group starting with $N > 1$ contributes k bordered rows to group G_1 - hence we deduce the smallest N value of the N-rep groups. Similarly we deduce the second smallest N value and repeat until we obtain the value of $N_1 = max$. In Example 4.1, $n = 17$, $k = 2$ and $T[11]$ is the end of the last N-rep group. Given 6 bordered rows remaining, and 2 N-rep groups, we see that the first N-rep group is in group G_3, and $max = 3$. The rotation of row 6 in G_3, shows that the first row in G_1, row 14, must have $N_2 = 3$; and so on.

2. Determine the length of each Z suffix in the first k unbordered rows of each N-rep group – Algorithm 6.

3. Implement the Last First mapping which uses $IndexRepN_2$ from Step 1, and the array of suffix lengths determined in Step 2 as a look-up table – Algorithm 7.

Algorithm 6 is a linear preprocessing step, bound by constant k times the length of the longest Z suffix of each Lyndon seed; Algorithm 7 processes each bit in the transform once with constant work:

Claim 4.3. *Procedure* COMPUTE-LF$_{rep}(T, Z, k, n)$) *can be implemented in linear time.*

The reasoning for correctness is similar to that in Claim 3.8, although the Z suffix of a block pair is obtained from a look-up table, and the current least unselected prefix is from k bordered or unbordered rows.

Claim 4.4. *Procedure* COMPUTE-LF$_{rep}(T, Z, k, n)$) *correctly recovers the input.*

5. Bijectivity

To complete the theory on the new binary transforms, we will establish that they are bijective, that is, if a string $\boldsymbol{t} = B$-BWT$(\boldsymbol{x})$ for an input B-word $\boldsymbol{x}$, which is specified to be type repetition or primitive, then the appropriate inversion method applied to $\boldsymbol{t}$ correctly recovers $\boldsymbol{x}$.

First we establish bijectivity for the repetition case for a given exponent of repetition k, and since the primitive case follows similarly, we will see that bijectivity holds for a fixed value of $k \geq 1$.

Lemma 5.1. *Suppose that $\boldsymbol{u}$, $\boldsymbol{v}$ are both B-rep words with exponent of repetition k. If $\boldsymbol{u} \neq \boldsymbol{v}$ then B-BWT$(\boldsymbol{u}) \neq B$-BWT$(\boldsymbol{v})$.*

Algorithm 6 Compute the Z suffixes for a transform T based on repetition.

```
procedure COMPUTE-Z_rep(T, k, n)
    Z[1...n];                 ▷ declare an array initialized to 0
    i := 1;                   ▷ i indexes the transform
    i' := 1;                  ▷ i' is the start of an N-rep group

    while i < n do
        N := 0;               ▷ N tracks how many 1's are detected
        k' := k;              ▷ k' tracks how many 1's are left to be detected

        while N < k do        ▷ find length of the first k Z suffixes in an N-rep group
            ℓ := i';          ▷ ℓ indexes the array Z

            for j = 1 to k' do        ▷ process suffixes from right to left
                while T[ℓ] is "marked" do ℓ := ℓ + 1
                end while

                if T[i] = 0 then Z[ℓ] := Z[ℓ] + 1
                else "mark" T[ℓ]; N := N + 1
                end if
                i := i + 1; ℓ := ℓ + 1
            end for
            k' := k − N
        end while

        while T[i] = 1 do i := i+1 ▷ skip bordered rows not in N-rep groups
        end while

        i' := i
    end while

    return Z
end procedure
```

Algorithm 7 Compute the Last First Mapping for a transform T based on repetition.

procedure COMPUTE-LF$_{rep}(T, Z, k, n)$
$currG := 1$; ▷ size of current group
$prevG := max$; ▷ size of previous group obtained from $IndexRepN_2$
$p := 10^z$; ▷ initialize the pattern for the matching process;
▷ z is obtained from a linear scan of $Z[1 \dots k]$,
▷ from Algorithm 6, giving the shortest suffix

while $|p| < n$ **do** ▷ iterate until n characters have been recovered

$t := IndexRepN_2[currG, prevG]$
▷ obtain index of current group in array M from the array $IndexRepN_2$

if $T[t] = 1$ **then** ▷ the last bit of the current rotation in M is 1
$p := 1 \circ p$; ▷ concatenate the suffix bit 1 to the start of p
$IndexRepN_2[currG, prevG] := IndexRepN_2[currG, prevG] + 1$;
$currG := currG + 1$ ▷ increment the group size

else ▷ current rotation in M ends 0 and belongs to an N-rep group
if $t = 1$ **then** "add marker to start of p" ▷ start of input $\boldsymbol{x}$
end if
$p := 10^{Z[t]} \circ p$; ▷ concatenate a Z block of size $Z[t]$ to the start of p
$IndexRepN_2[currG, prevG] := IndexRepN_2[currG, prevG] + 1$
$prevG := currG$;
$currG := 1$ ▷ the prefix 10 indicates a rotation in group G_1 (Corollary 2.12)
end if

end while

return $LF_{rep}(\boldsymbol{x})$ ▷ the input $\boldsymbol{x}$ is given by the rotation of p starting with the marker
end procedure

Proof. Suppose that B-BWT($\boldsymbol{u}$) = B-BWT($\boldsymbol{v}$) = $\boldsymbol{t}$, then $|\boldsymbol{u}| = |\boldsymbol{v}| = |\boldsymbol{t}| = n$. Denote the B-BWT matrices of $\boldsymbol{u}$, $\boldsymbol{v}$ as $M(\boldsymbol{u})$ and $M(\boldsymbol{v})$ respectively. Lemma 2.15 shows that the transform $\boldsymbol{t}$ partitions into substrings of two distinct types: those whose elements are the suffixes of N-rep groups, and those whose elements are the suffixes of bordered rotations formed from splitting N blocks.

Consider a substring $\boldsymbol{s}$ of $\boldsymbol{t}$ corresponding to an N-rep group, which we assume forms rows i to j in $M(\boldsymbol{u})$ – the first N-rep group starts at $\boldsymbol{t}[1]$ and $M(\boldsymbol{u})[1, n]$. Every such substring $\boldsymbol{s}$ starts with 0^k and ends with the k^{th} occurrence of a 1 in $\boldsymbol{s}$. So the first k rows corresponding to this substring $\boldsymbol{s}$, $i, i+1, \ldots, i+k-1$, in $M(\boldsymbol{u})$ must all end with 0, while the suffix bits of the following k rows in $M(\boldsymbol{u})$ (which must exist), corresponding to $\boldsymbol{s}$, are the $(n-1)^{th}$ elements in the rows $i, i+1, \ldots, i+k-1$ in $M(\boldsymbol{u})$. Similarly for the next $k-d$ rows, where d is the number of 1's in the previous k suffix bits. Therefore, there is a 1-1 correspondence between $\boldsymbol{s}$ and the Z block suffixes of rows $i, i+1, \ldots, i+k-1$ in $M(\boldsymbol{u})$. Hence this must also hold for $M(\boldsymbol{v})$. Furthermore, since this is the case for all N-rep groups, the Z values in $\hat{Z}(\boldsymbol{u})$ and $\hat{Z}(\boldsymbol{v})$ must be the same – but not necessarily their order in $\boldsymbol{u}$ and $\boldsymbol{v}$.

Applying Step 1 of the inversion method, we first deduce the N value of the first N-rep group ($N \geq 1$) and subsequently find the other N values for the remaining N-rep groups. Hence we know the N values of $\hat{N}(\boldsymbol{u})$ along with their repetition factor, which similarly must be the same for $\hat{N}(\boldsymbol{v})$, but as above, not necessarily their order. It follows that we also know each of the NZ pairs (from clockwise rotations of N suffixes to concatenate with Z prefixes) which must be the same in both $\boldsymbol{u}$ and $\boldsymbol{v}$ – but once again, not necessarily their order. So we proceed to show that the order of these block pairs must be the same in both $\boldsymbol{u}$ and $\boldsymbol{v}$.

All this implies that both $\boldsymbol{u}$ and $\boldsymbol{v}$ have the form $1^{max} \cdots 0^{Z_r}$. This last Z block suffix is used for starting the Last First mapping, hence the initialized pattern p is the same in both $\boldsymbol{u}$ and $\boldsymbol{v}$.

It also follows that since the N-rep groups are the same in both matrices, then so to are multiples of k bordered rows preceding or following these groups. Hence we need to show that the ordering of a bordered group starting with some N_i, and an N-rep group also starting with N_i, is the same in both matrices.

From Step 1 of the inversion method, the entries in the array $IndexRepN_2$ will be the same for both $\boldsymbol{u}$ and $\boldsymbol{v}$. The Last First mapping specifies a sequence of rows visited in the matrices, where the suffix (0 or 1 bit) of a row is the same in both $M(\boldsymbol{u})$ and $M(\boldsymbol{v})$. Hence the same sequence of N values (including split N's) is determined for both $\boldsymbol{u}$ and $\boldsymbol{v}$. Furthermore, the look-up table given by array $Z[1 \ldots n]$ in Algorithm 6 shows that the i^{th} N in both $\boldsymbol{u}$ and $\boldsymbol{v}$ will have the same adjacent Z values – that is, the sequence of block pairs is the same in $\boldsymbol{u}$ and $\boldsymbol{v}$. □

Finally, since a similar argument holds for the primitive case, we can state:

Lemma 5.2. *Suppose that $\boldsymbol{u}$, $\boldsymbol{v}$ are B-prim words. If $\boldsymbol{u} \neq \boldsymbol{v}$ then B-BWT($\boldsymbol{u}$) $\neq$ B-BWT($\boldsymbol{v}$).*

Table 1: Improvement gained in the number of blocks for the Rouen Transform versus the Burrows-Wheeler Transform for all the B-words of length from 4 to 20. For instance there are 30 B-words of length 19 whose Rouen Transform has 8 blocks less than the original Burrows-Wheeler Transform.

	Number of blocks													
	8	6	4	2	0	-2	-4	-6	-8	-10	-12	-14	-16	Total
4				1	2									3
5				2	3	1								6
6				3	6									9
7				7	9	1	1							18
8				13	13	3	1							30
9			2	22	19	12		1						56
10			5	35	42	12	5							99
11			14	61	76	21	12	1	1					186
12		1	29	109	115	62	14	4	1					335
13		4	59	201	219	96	30	19	1	1				630
14		9	133	341	386	182	81	23	6					1161
15		29	260	629	685	347	161	50	20		1			2182
16		74	506	1140	1141	662	300	132	19	5	1			4080
17	2	169	1019	2102	2117	1257	628	236	51	27	1	1		7710
18	7	396	1973	3854	4054	2374	1201	487	148	31	7			14532
19	30	882	3865	7195	7308	4492	2447	940	331	74	28	1	1	27594
20	105	1897	7259	13197	13507	8545	4750	1938	627	226	29	6	1	52377

Interestingly, the B-prim word 111001101000 (with repetition exponent 1) and the B-rep word 110101100100 (with exponent 2), both have the same transform, namely 000110010111. This shows the necessity of specifying the type of B-word, the exponent, and hence the inversion method required: Algorithm 5 or Algorithm 7.

Hence like human twins, these transforms are very similar but distinct: with repetition exponent 1 (although strictly > 1), a B-rep word would require that $\hat{Z}$ satisfies Lyndon properties, while $\hat{Z}$ is unrestricted for B-prim words. Hence B-prim words can be viewed as a special case of B-rep words.

6. Experimentation

We generated all the B-words of length from 4 to 20. For each of them we computed both the Rouen (Block Order) Transform and the Burrows-Wheeler Transform; furthermore, we computed the improvement gained in the number of blocks for the Rouen Transform versus the Burrows-Wheeler Transform (see Table 1). These results show that in most cases the two transforms have the same number of blocks; however, in most of the remaining cases the Rouen Transform generates less blocks than the Burrows-Wheeler Transform. This means that a proper decomposition of any given binary word into B-words could lead to a better compression with the Rouen Transform than with the

Burrows-Wheeler Transform. These encouraging preliminary results show that further experiments need to be done for a thorough classification.

7. Conclusion

We have introduced a class of binary Burrows-Wheeler type transforms for binary block order - we believe this is the first approach for transforms designed specifically for binary inputs. We have described two types of B-words: primitive B-prim words and repetition B-rep words; furthermore linear factoring of binary strings into B-words is established.

Computing both the B-BWT transforms and inverses are shown to be linear and bijective; the transform applies suffix array techniques while the inversion is based on combinatorial analysis and indexing. In addition to the theoretical contribution, we have demonstrated that it may be worthwhile in practice to implement the Rouen Transform as preprocessing for compression.

An obvious quest for future research is to devise a fully bijective linear transform for binary block order over arbitrary inputs. If the given string is not a B-word, then it could be factored into these patterned words, however the multi-word approach by Kufleitner [Kuf09] is not readily applicable as it requires sorting the elements of the multi-transform into lexorder in the first column of the matrix M. Even avoiding specifying the frequency exponent k, when $\hat{N}$ is a repetition, with its overhead of $\log n$, seems to be a challenge.

Our ultimate interest is to consider degrees of data clustering achievable by different transforms which are derived from a variety of total orders on suitable alphabets. The integer encoding of strings $\hat{\boldsymbol{x}}$ gives rise to 32 orders by prioritizing N or Z values and using $<$ or $>$ for the ordering, which includes binary type order [DD03] – as a next step we suggest investigating other new binary transforms to be defined using these orders for the BWT matrices.

In addition to thorough experimental analysis, for instance on genome-based repetitions, we suggest comparing efficiencies obtained when performing a transform on ASCII characters directly, or, converting ASCII characters to binary and then performing a binary transform.

We thus propose that a valuable line of enquiry is to consider binary BWT transforms in the context of bit-optimal compression methods: the goal of the bit-optimal parsing problem is to minimize the length in bits of an encoded text [Lan13].

8. Acknowledgements

We thank the anonymous reviewers for carefully reading the manuscript and their helpful comments.

9. References

[ABM08] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

[AT05] J. Abel and W. Teahan. Universal text preprocessing for data compression. *IEEE Trans. Comput.*, 54(5):497–507, 2005.

[BCRS13] M. J. Bauer, A. J. Cox, G. Rosone, and M. Sciortino. Lightweight LCP construction for next-generation sequencing datasets. *CoRR abs/1305.0160*, 2013.

[BW94] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[CDP05] M. Crochemore, J. Désarménien, and D. Perrin. A note on the Burrows-Wheeler transformation. *Theoret. Comput. Sci.*, 332(1-3):567–572, 2005.

[CFL58] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus IV — The quotient groups of the lower central series. *Ann. Math.*, 68:81–95, 1958.

[CGKL13] M. Crochemore, R. Grossi, J. Kärkkäinen, and G. M. Landau. A constant-space comparison-based algorithm for computing the Burrows-Wheeler transform. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 74–82, 2013.

[CHL07] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[CT98] B. Chapin and S. R. Tate. Higher compression from the Burrows–Wheeler transform by modified sorting. In *Proc.1998 Data Compression Conf. (DCC)*, page 532, 1998.

[Day11] D. E. Daykin. Algorithms for the Lyndon unique maximal factorization. *J. Combin. Math. Combin. Comput.*, 77:65–74, 2011.

[DD96] T. -N. Danh and D. E. Daykin. The structure of V-order for integer vectors. In A. J. W. Hilton, editor, *Congressus Numerantium*, volume 113, pages 43–53. Utilitas Mat. Pub. Inc., Winnipeg, Canada, 1996.

[DD03] D. E. Daykin and J. W. Daykin. Lyndon-like and V-order factorizations of strings. *J. Discrete Algorithms*, 1:357–365, 2003.

[DS14] J. W. Daykin and W. F. Smyth. A bijective variant of the Burrows-Wheeler transform using V-order. *Theoret. Comput. Sci.*, 531:77–89, 2014.

[Duv83] J.-P. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[DW14] J. W. Daykin and B. Watson. A text transformation scheme for degenerate strings. In Costas S. Iliopoulos and Alessio Langiu, editors, *2nd International Conference on Algorithms for Big Data (ICABD2014)*, number 1146 in CEUR-WS Proceedings, pages 23–29, Aachen, 2014.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science, (FOCS 2000)*, pages 390–398, 2000.

[GS12] J. Y. Gil and D. A. Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012.

[KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 200–210, 2003.

[Kär07] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoret. Comput. Sci.*, 387(3):249–257, 2007.

[KKP12] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Slashing the time for BWT inversion. In *Proc. Data Compression Conference (DCC)*, pages 99–108, 2012.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th Internat. Conf. Automata, Languages & Programming*, pages 943–955, 2003.

[KSB06] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J.ACM*, 53(6):918–936, 2006.

[Kuf09] M. Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. Stringology*, pages 65–79, 2009.

[Lan13] A. Langiu. On parsing optimality for dictionary-based text compression - the Zip case. *J. Discrete Algorithms*, 20:65–70, 2013.

[Lot83] M. Lothaire. *Combinatorics on Words*. 2nd Edition, Addison-Wesley, Reading, MA (1983); Cambridge University Press, Cambridge (1997). Addison-Wesley, 1983.

[MRRS05] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows Wheeler transform and applications to sequence comparison and data compression. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 178–189, 2005.

[NZC09] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proc. 2009 Data Compression Conf. (DCC)*, pages 193–202, 2009.

[SLLM09] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. A four-stage algorithm for updating a Burrows-Wheeler transform. *Theoret. Comput. Sci.*, 410(43):4350–4359, 2009.

[Smy03] B. Smyth. *Computing Patterns in Strings.* ACM Press Bks. Pearson/Addison-Wesley, 2003.